COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY · STANFORD, CA 94305-4055

THE ANNA PACKAGE SPECIFICATION ANALYZER USER'S GUIDE

Walter Mann

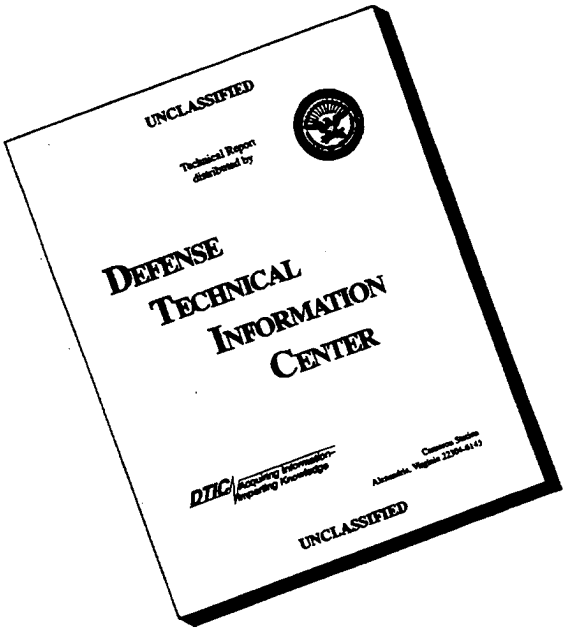
Technical Note: **CSL-TN-93-390**

(Program Analysis and Verification Group Report No. 60)

January 1993

This research has been supported by the Defense Advanced Research Projects Agency/Information Systems Technology Office under the Office of Naval Research Contract N00039-91-C-0162.

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

The Anna Package Specification Analyzer User's Guide

Walter Mann

Technical Note: CSL-TN-93-390
Program Analysis and Verification Group Report No. 60

January 1993

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

The Anna Package Specification Analyzer is a tool used in developing Ada package specifications annotated with Anna language constructs. The tool constructs a symbolic representation of a package specification, and models different states of that package. Using deductive reasoning on the model, it answers questions about those states, and, if the specification is complete enough, simulates by symbolic execution how an implementation of the package which satisfies the specification would behave, even if no such implementation exists.

In consequence, the user has greater confidence in the resulting specification; inconsistencies detected later by run-time checking tools are more likely due to errors in the implementation, rather than in the formal specification.

Key Words and Phrases: formal specification, Ada, Anna, formal analysis, symbolic execution

Contents

Chapter 1

Introduction

Anna [1, 2, 3] is a specification language for sequential Ada programs which allows users to specify formally the intended behavior of Ada software. In particular, the visible part of an Ada package may be formally specified by annotations on types and operations declared in it, and by axioms constraining the overall behavior of the package. Any implementation (body) provided for this package must satisfy its constraints.

The Anna Package Specification Analyzer (referred to in this paper as the Analyzer for brevity) is a tool used in developing Ada package specifications annotated with Anna language constructs. The tool constructs a symbolic representation of a package specification, and models different states of that package. Using deductive reasoning on the model, it answers questions about those states, and, if the specification is complete enough, simulates by symbolic execution how an implementation of the package which satisfies the specification would behave, even if no such implementation exists.

The tool provides an interactive environment which simulates how an actual computation would use the package. The user acts like a driver program, executing the package operations symbolically and examining the results. Inconsistencies may arise in the course of analysis: the package may not behave as the user intended, yielding for instance an unexpected result for a function call. The user may use the Analyzer to deduce which annotations implied the incorrect result, and hence why the specification is unacceptable.

For example, consider the fragment of a file Input/Output package in Figure 1.1. The user has specified subprograms and in-state and out-state conditions for them. At some point, the user may wonder whether the following case is constrained as expected by the specification: two files (where a file is some abstract type) are opened, both to the same external file, and one is deleted—what is the status of the other file?

With the Analyzer the user may test this case by performing the operations and then examining the resulting state. The following short session models the desired state and tests the case. User input is shown in **boldface**, and Analyzer output in Roman.

First the file containing the package of Figure 1.1 is loaded, and the Analyzer builds a model of the specification. This model is called the *computation state* in that it simulates an embedding of the package in an environment where the user can examine and change the state of the environment. Interactions with the package transform the computation state as new information arises.

Then two variables of type File_Type are declared interactively from within the environment of the Analyzer. We use these variables to represent symbolic entities which we may manipulate in subprogram calls.

```
package IO is
    type File_Type is private;
    Status_Error : exception;

    function Is_Open( F : File_Type ) return Boolean;

--| axiom not Is_Open( File_Type'Initial );

    procedure Open( F : in out File_Type;
                    Name : in String );
--| where Is_Open(F) => raise Status_Error,
--|       out( Is_Open(F) );

    procedure Delete( F : File_Type );
--| where not Is_Open(F) => raise Status_Error,
--|       out( not Is_Open(F) );

    :
end IO;
```

Figure 1.1: Fragment of an Input/Output Package

```
> p (Load package specifications)
--> io.anna

> o (Declare objects)
--> F1, F2 : File_Type;
-->
```

The user enters the operations which would be executed to reach the desired state: two Open operations, followed by the Delete operation.

```
> x (Execute subprograms)
--> Open( F1, "myfile" );
--> Open( F2, "myfile" );
--> Delete( F1 );
-->
```

Each call results in the symbolic execution of the given subprogram call. In this case symbolic execution consists of deducing that the in-conditions of the operations are satisfied in the current state, and then asserting the out-conditions.

First the Analyzer tests whether the in-state conditions of the subprogram call are satisfied by knowledge in the current computation state. In both calls to Open here the package axiom

```
--| axiom not Is_Open( File_Type'Initial );
```

means that every object of type File is initially closed, so the in-condition of the first Open call is satisfied.

If the in-conditions are satisfied, the state of computation is updated such that the out-state conditions

are TRUE in the new state. The execution of the first Open call makes the expression

Is_Open(F1)

TRUE after execution of the call. If this expression had not been TRUE when the call to Delete was symbolically executed, the propagation annotation of subprogram Delete would have been violated, and the user would be notified that an exception would occur in this case.

Once the state in question has been reached, the user asks what the value of function Is_Open would be if it were called with the undeleted file.

```
> v (Value of expression)
--> Is_Open(F2);
-->
TRUE
```

From the formal specifications, the Analyzer has deduced that F2 would still be open in this state of computation, even when its external file has been deleted. If this was not the intended behavior of an implementation, the user may wish to refine the annotations to reflect the intended behavior.

In consequence, the user has greater confidence in the resulting specification; inconsistencies detected later by run-time checking tools such as [4] are more likely due to errors in the implementation, rather than in the formal specification.

This tool has been implemented on Unix variants using the Verdix Ada Development System (VADS) on Sun-3 and Sun-4 workstations and a Sequent Symmetry multiprocessor[1].

Chapter 2 gives an overview of the concepts of specification analysis. Chapter 3 discusses the concept of State, which is crucial to Specification Analysis; it also includes a walk-through introductory session for getting started with the tool. Chapter 4 lists all commands available, and how to use them. Chapter 5 discusses the implicit knowledge and assumptions made by the Analyzer. A complete session of specification analysis is given in Appendix A.

[1] Unix is a registered trademark of AT&T Bell Laboratories. Verdix and VADS are registered trademarks of VERDIX Corporation. Sun is a registered trademark of Sun Microsystems, Inc. Sequent and Sequent Symmetry are registered trademarks of Sequent Computer Systems, Inc.

Chapter 2

Specification Analysis—Overview

Specification analysis refers to tools and methods for analysis and debugging of formal specifications; formal specifications are considered as entities independent of implementations. That is, specification analysis may be used before an implementation of the specification has been written, so that errors and ambiguities are uncovered early in the software life-cycle.

A *specification analyzer* is an interactive tool which creates an environment. The environment includes a *computation state* which undergoes incremental transitions. The state models the specification, variables declared interactively, and some predefined knowledge. The environment also supports several classes of commands; the two most important of these are manipulation and query commands.

State *manipulation* operations allow the user to change the model by extending or restricting it, and thereby simulate how an implementation satisfying the constraints would function. For example, an Anna subprogram may be symbolically executed, meaning that the model is changed such that future responses will behave as though the actual subprogram (as implied by its formal description) were executed.

State *query* operations let the user examine the knowledge implied by the state. For example, the user may ask whether a particular Anna expression holds TRUE in the current state.

Figure 2.1 illustrates how a user interacts with the Analyzer using manipulation and query operations.

At this point, a complete example might be helpful. Section 3.1 includes a session using the Analyzer which introduces the computation state, while Appendix A is a complete scenario of a specification debugging session.

The original work in specification analysis for Anna was done by Neff [5], who describes an early tool, while [6] describes how the Ada/Anna constructs that comprise a specification are modeled in the tool.

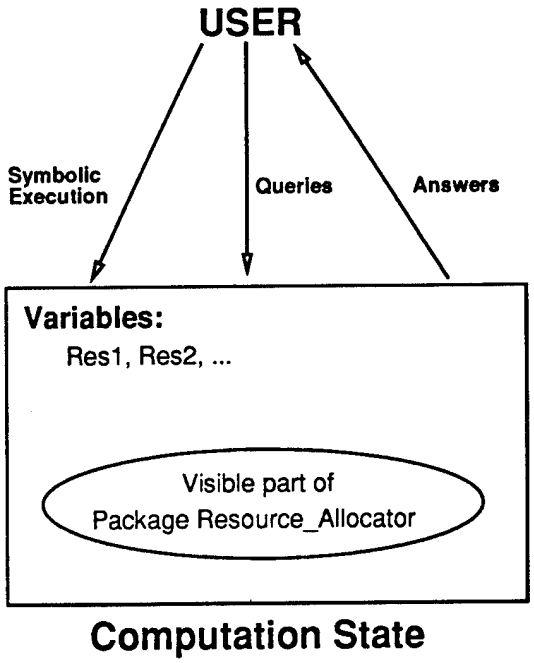


Figure 2.1: User Interaction With the Analyzer

Chapter 3

Computation State

The basic environment provided by the Analyzer is the *computation state*. This state can best be thought of as representing the knowledge given in the specification, both by specific Anna statements, and by all that they imply. Another way to think of the computation state is as a logic state—a set of boolean statements describing what is true of the specification.

The state undergoes various *transformations* as analysis progresses. These transformations consist of adding new information to the original state; no information is thrown away, as new additions to the state may refer to values in the old state. For example, suppose a variable has a certain value in a computation state. We want to change the state so that the variable has a new value which is a function of the old value. This state transformation is modeled by adding information relating the variable's new value in terms of its old value.

States are indexed by a natural number, initially zero (0). Each transformation increments this number. So the index number of a state indicates how many transformations have occurred since its initial state.

A specification analysis session begins with the initial computation state, which consists of predefined knowledge of the Ada language (such as information about predefined types: integers, characters, etc.). Each package specification loaded causes a state transformation, which adds the knowledge acquired from the specification to the old state; the index number is incremented each time.

Subsequent transformations are caused by other Analyzer commands, namely: symbolic execution of a subprogram, interactive declaration of objects, and explicit state updates.

3.1 Getting Started

The following Analyzer session introduces the computation state. This section assumes that the tool has been properly installed along with the rest of the Anna toolset, and has been invoked [1].

At this point the user sits in the environment called the Computation State. Initially the Analyzer has no knowledge of any package specifications, but it does have predefined information found in package Standard, such as knowledge of types Integer, Boolean, Character, etc., as well as the predefined operations on these types.

The following session fragment is intended purely to familiarize the user with the basic concepts of the computation state; it should not be considered a typical session. The characters after the "> " prompt are single character commands entered by the user, and the Anna code following the "--> " prompt is also entered by the user. Note that an empty line must follow Anna code to indicate the end of input.

First we declare two variables. This causes a transformation of the computation state, such that the new information is added. That is, the computation state number changes from 0 to 1. Then we ask about the value of an expression involving the variables:

[1] See [7] for details in installing and invoking Anna tools on your system

```
> o (Declare objects)
Enter Code (<CR> to end):
--> I, J : Integer := 3;
--> B : Boolean := not False;
-->

> v (Value of expression)
Enter Code (<CR> to end):
--> I * J + Boolean'Pos(B);
-->
10
```

Using the 'S' command we can cause another computation state transformation such that the variables have new values, and then test those values.

```
> S (Update state)
Enter Code (<CR> to end):
--> I = in I + 1,
--> J < 0;
-->

> v (Value of expression)
Enter Code (<CR> to end):
--> I, J, B;
-->
4

Unable to deduce a value for this expression.

TRUE
```

The new value for I is 4 because in the state update we said its value in the new state was equal to its value in the old state (expressed in Anna as **in** I) plus 1. We cannot deduce a value for J because the state update did not contain enough information to prove a specific value, only that it is less than 0. Finally, since we did not mention B in the update, the Analyzer assumes its value does not change, and so it is still TRUE.

We can also use the Display command to see an explicit representation of the computation state as a list of Anna expressions. It also displays the current computation state number. Note that the "current" value of variable I is represented as "i'2" and is shown in terms of the value of I in the previous state. Computation state 1 is the subset of computation state 2 consisting of its first three statements.

```
> d (Display computation state)
Current Computation State: 2
---------------------------

i'1 = 3
j'1 = 3
b'1 = TRUE
i'2 = i'1 + 1
j'2 < 0
```

3.2 Computation State vs. Package State

Some confusion may arise over the relationship between the state of computation and the Anna definition of *package state*. In Ada, packages can simulate two different abstractions: data types and state machines. A single package can in fact use both abstractions, but for explanatory purposes we will discuss only those packages which are "pure" abstractions of either kind.

An example of an abstract data type package is a mathematics package, such as the complex number package given in Figure 3.1; it defines a type Complex_Number and operations on it such as +, −, etc. A package such as this requires no internal memory; it simply defines routines, the resulting values of which are pure functions of their input. The state (set of internal data structures) of this kind of package does not change. We say that such a package requires only a *trivial state.*

An example of an abstract state machine package is a resource allocator, the interface of which has operations like Resource_Available, Allocate, and Deallocate; an example of such a package is given in Figure 3.2. Resource_Available may return TRUE initially, but FALSE if another unit has called Allocate; in a sense, the package must "remember" that Allocate has been called. Such a history-dependent package requires a *state*, internal memory (such as data constructs in its body), to respond as intended.

In Anna all packages, whether or not they require a non-trivial state, are extended to be full types. A package type defines a domain, which consists of all possible states of the package (the state type of a trivial state package therefore has exactly one element in its domain). In Anna, a package state is denoted by the list of calls to subprograms of the package since its initial state which were required to reach the state.

For instance, in the Resources package, we can describe the state achieved by starting in the current package state, executing Allocate, and then executing Deallocate, by the Anna expression:

```
Resources'State[Allocate(Mag_Tape); Deallocate(Mag_Tape)]
```

Then we can express that any state of the allocator package is equal to the state achieved by executing Allocate and then Deallocate for the Mag_Tape resource, with the following axiom:

```
--| axiom
--|     Resources'State[Allocate(Mag_Tape); Deallocate(Mag_Tape)] =
--|     Resources'State;
```

In the Complex_Number package, every list of subprogram calls describes the same state. The sequence of subprogram calls previously made to the package have no effect on the current subprogram call.

Now, how is the package state related to the computation state created by the Analyzer? Conceptually, packages under analysis are declared "inside" the computation state, and so the current state of a package is a subset of the current state of computation. When a particular package state changes, for instance as a result of symbolic execution of one of its subprograms, the computation state is therefore transformed, since it is a superset of the package state. However, not every computation state transformation will change a package state; updating the value of a variable will change the computation state, but will not change any package states.

In Figure 2.1, changing a variable such as Res1 causes a computation state transformation, which increments the computation state number, but which does not affect the state of the Resource_Allocator package.

Executing a subprogram, say Allocate, in the Resource_Allocator transforms the state of that package, which also means that the computation state has transformed, so again the computation state number is incremented.

```
package Complex_Number_Package is
    type Complex_Number is private;

    function "+"( C1, C2 : Complex_Number ) return Complex_Number;
    function "-"( C1, C2 : Complex_Number ) return Complex_Number;
    :
end Complex_Number_Package;
```

Figure 3.1: Fragment of a Complex Number Package

```
package Resources is
    type Resource is ( Printer1, Printer2, Mag_Tape );

    function Resource_Available( R : Resource ) return Boolean;

    procedure Allocate( R : Resource );

    procedure Deallocate( R : Resource );
    :
end Resources;
```

Figure 3.2: Fragment of a Resource Allocation Package

3.3 Variables and Functions

Variables may be declared interactively at any point during specification analysis, and are used to store values. Their values may change—the value of a variable may be different in different states of a computation. So we can say that a variable has a value *with respect to* a specific computation state. During analysis, the current value of a variable may change, but its value with respect to a specific computation state always remains the same.

The value of a variable with respect to a computation state is expressed in the Display Computation State command by displaying the computation state associated with every variable instance. For example, "v'3" refers to the value of variable v in computation state 3.

Variables may also be declared in a package specification. However, such variables are not considered to be part of the package state, since their values may be changed by threads of control outside the package

boundary. By Ada rules, package variables are considered to exist in the scope declaring the package[8, §8.2]. In specification analysis, this outer level is the computation state environment itself, and package variables may be thought of as existing at the same "level" as interactively declared variables, and are treated the same way—they are subscripted by the computation state number.

In Specification Analysis, functions are always associated with some package specification. Functions have a value with respect to the state of the package in which they were declared. This may correspond to several states of the computation, however a user may only refer to the value of a function in terms of package state, using the relative function call and successor state syntax of Anna[2, §7.7.3].

Chapter 4

Commands

There are five classes of commands available: loading, manipulating, querying, logging/scripting, and other miscellaneous commands. In the default line-oriented interactive mode or the Analyzer, all commands are single characters, not requiring a carriage return; some require further input such as a file name, Anna expressions, etc. In this chapter we first describe the input required by the Analyzer, and then discuss the commands themselves.

4.1 Kinds of Input

The Analyzer takes three different kinds of input: commands, file names, and Anna code. Each is discussed below.

Commands. In line-oriented mode, the command prompt "> " requires a single character response with no carriage return. In windowing mode, commands are listed in pull-down menus. The user is prompted for any further required input.

File Names. Commands requiring a file as input give the prompt "--> ". The file name should be entered, followed by a carriage return. The full file name (with extensions such as ".anna") may or may not be required; see specific commands for details.

Anna Code. Anna code is also prompted for by "--> ". All Anna code input requires a terminating semi-colon ';' to indicate the syntactic end of the expression. Further, the Analyzer requires that a blank line (with no spaces or tabs) be entered after the last line of Anna input, to indicate the end of input. Some commands accept multiple Anna expressions; these expressions should be separated by commas, and terminated by a semi-colon.

For commands which require File Names or Anna Code, if no input is given the command is aborted with no change to the computation state.

Occurrences of a variable in an Anna input expression refer to the value of the variable with respect to the current (numbered) computation state, except those which are modified by **in**; those occurrences refer to the value of the variable in the previous computation state. Function calls in an expression require the full package name prefix, for instance:

```
Stack.Pop
```

unless the package was the last one loaded (specifically, when analyzing a single file, the package name need not be given). In Ada semantic terms, the environment acts as though all packages under analysis are **with**ed, and the last one loaded is also **use**d, by the computation state.

A function call may be given relative to a specific package state, for instance:

```
Stack'Initial.Pop
Stack_Object[Push(0)].Pop
```

or without any reference to a state:

```
Pop
```

The last example refers to the value of the function in the current package state.

Declarations and subprogram executions entered interactively which refer to elements declared in virtual text must follow the Anna semantics rule that they be virtual text themselves; start each such input line with the virtual text indicator "--:", following the prompt.

4.2 Loading Commands

Loading commands determine which specifications are being analyzed; specifications are loaded from external files. The standard sequence for loading a series of files is: start with the initialization command, followed by one or more package load commands. This sequence may be repeated later in the session to analyze another set of specifications.

4.2.1 Initialize — 'i'

This command reinitializes the computation state to state 0, with no packages loaded and no interactive variables declared. If, after a session of analysis, the user wishes to analyze a different set of packages or to reload modified versions of the packages being analyzed, the 'i' command must be issued before the next set of loads.

4.2.2 Load Packages — 'p'

This command prompts the user for the name of a file containing one or more Anna package specifications, and loads the file packages into the Analyzer. If the file name ends with the extension ".anna" that extension may be omitted.

If the file is successfully loaded, the current state of each package is its Initial state. Queries about the state of any package which are deducible purely from the object and type annotations, axioms, and function result annotations can then be made.

Example:

```
> p (Load package specifications)
Enter File Name:
--> mypkg.anna
```

4.2.3 Reset to Initial States — 'r'

This takes no further input. It resets the computation state to its initial value plus the initial states of all packages given in the last set of loads. That is, it acts as though the computation state were reinitialized and any files given in the last set of 'p' commands were reloaded.

The Reset command is useful if the user wants to start the session over again quickly, perhaps because he has made an unrecoverable input error in the last session.

4.3 Querying Commands

This group of commands allows the user to examine and test the computation state. The Analyzer includes a theorem-prover which uses deduction on the computation state to derive its implications.

4.3.1 Value Search — 'v'

The user is prompted for a list of Anna expressions. The type of each expression must be discrete: specifically, an integer, enumeration, or character type, subtype, or derived type. As a special case, string values are also supported, but indexing, slices, and catenation on strings are not supported.

For each expression, the state is examined and a set of Ada literals is returned.

If the state is consistent, this set consists of the literals in the equivalence class of the expression. It will always be either a singleton set or the empty set, due to the uniqueness property of Ada literals (that unique literals always have unique values).

The value command is useful in finding the value of an observer function of a package without actually changing the state. The result returned is the value of the function if it were executed with the Execute Subprogram command. Note however that in-conditions of a function are not currently checked when using the Value Search command; these conditions are assumed to be satisfied.

Expressions containing quantified subexpressions are evaluated using the Objects Define Domain method given in Appendix C.

Example:

```
> v (Value of expression)
--> Cardinality(My_Set);
-->

5
```

If the computation state is inconsistent and the Single Value Search option is set to FALSE (see Section 4.6.1), the set of literals produced by this command consists of the set union of the value of the expression in the *consistent substates* of the state. Recall that the state can be described as a set of boolean statements. A consistent substate is a subset of this set which is consistent. Sometimes a state is inconsistent because the value of an expression is implied to be equal to more than one literal; if so, all such literals are given. Hopefully, this will aid in debugging the inconsistency.

Example:

```
> v (Value of expression)
--> Cardinality(My_Set);
-->

2
1
```

If the Single Value Search option is TRUE, only the first literal found in the equivalence class of the expression is printed.

4.3.2 Boolean Query — 'q'

The user is prompted for a list of Anna boolean expressions. For each expression, TRUE, FALSE, UNKNOWN, or INCONSISTENT is returned, indicating the truth value of the expression with respect to the computation state.

This command is a special version of the Value Search command for use with Boolean-typed expressions. There are two main differences between it and Value Search.

Firstly, the Single Value Search option applies only to Value Search; Boolean Query will always search for all values of the expression. If the state implies that the expression is TRUE only or FALSE only, the command will return TRUE or FALSE, respectively. If neither value is implied, UNKNOWN will be returned. If both values are implied, INCONSISTENT will be returned.

Secondly, the user has control over which method of evaluating Anna quantified expressions is used; in Value Search, the Objects Define Domain method is always used. Section 4.6.1 describes an option which changes the evaluation method, and Appendix C describes the methods.

A TRUE or FALSE answer can be accepted confidently, but UNKNOWN can mean either that not enough information exists to prove or disprove the queried statement, or that the theorem prover was not intelligent enough to find a proof or disproof.

For example, suppose we have declared two integer variables A and B, but their values are not known. The query:

```
A = B;
```

will return with UNKNOWN because we have not provided the Analyzer with enough information to determine if this statement is TRUE or FALSE.

However, suppose we made the query:

```
A + B = B + A;
```

In the *current implementation* of the Analyzer, this would also return UNKNOWN, even though it is TRUE for all values of A and B; the symmetry axiom of addition is not part of the implicit knowledge of the Analyzer (see Chapter 5 on Implicit Knowledge), and so the statement cannot be proved.

Example:

```
> q (Query state)
--> Push(I1,Push(I2,S)) = Push(I2,Push(I1,S)),
--> Length(Push(I1,S)) > 0;
-->

UNKNOWN
TRUE
```

4.3.3 State Consistency Test — 't'

This command takes no further input. It checks whether the current computation state is consistent. If so, it simply returns. If not, an inconsistent subset (not necessarily minimal) of the Anna expressions comprising the state is printed out. This command can be invoked automatically each time the state is translated; see the Options command.

Example:

```
> t (Test state consistency)
ERROR: State is inconsistent:
for all T : T_Type => 0 < T;
T1 = -5;
```

4.3.4 Display Computation State — 'd'

This command displays a representation of the computation state (except the implicit knowledge discussed in Section 5): axioms, function result annotations, and values of variables and function calls parameterized by specific computation states. The form of the display is a list of Anna expressions (with the exception of variables, as is described below). The expressions may not be exactly the same as the original text of annotations and state transformations, because all logical expressions are transformed into a canonical form called *disjunctive normal form* to aid theorem proving techniques. The current computation state (numerical) and each package's current state (list of subprograms executed since its initial state) are also shown.

Example:

```
> d (Display computation state)
Current Computation State: 1
Current Package State for stack: stack'Initial[push(0)]
-------------------------------

stack'Initial.length = 0
stack'Initial[push(0)].length = 1
stack'Initial[push(0)].top = 0
```

4.4 State Manipulation Commands

States can be manipulated in various ways, but all follow the same general principle. The computation state undergoes a transformation, such that the information comprising the transformation is made TRUE in the new state. The computation state number is incremented; state manipulation commands may also change package states—see specific commands for details. What happens to other information not referred to in the transformation is discussed in Section 5.2 on the Frame Axiom.

4.4.1 Declare Objects — 'o'

This command prompts the user for one or more declarations of objects (variables or constants) which are used to store values during analysis. Along with package variables, variables declared in this way reside outside all package states, and the frame axiom schema for variables applies to them. Initial values can be given to the objects, and they can be annotated with object constraints, which constrain their values in all states of the computation. All package states are unchanged after executing this command.

Example:

```
> o (Declare objects)
--> Even1 : Integer;
--> --| Even1 mod 2 = 0;
--> First_Char, Lower_Case_A : constant Character := 'a';
--> --: My_State_Obj : Stack'Type := Stack'Initial;
-->
```

4.4.2 Update Computation State — 'S'

This command allows the user to state explicitly a transformation of the computation state. The user is prompted for a list of Anna expressions, separated by commas and terminated by a semi-colon. This

causes a state transformation such that the expressions are TRUE in the new state. The package states are unchanged.

Example:

```
> S (Update state)
--> X = 0,
--> My_State_Obj = Stack'State;
-->
```

This command is mainly intended to update manually the values of interactively declared variables. To change the value of some set of variables, simply execute the command with a set of equality updates such as in the example above.

However, manual state updating may also be used to postulate how a package would react if new axioms were added to it. An axiom applies over all states of the package. This can be expressed by updating the computation state with an expression where all calls to a package subprogram are universally quantified over the state type of the package in which they were declared.

For example, using the Resource Allocation package of Section 3.2, suppose we want to examine the results of adding an axiom specifying that any state of the package is equivalent to a state achieved after allocating and then deallocating a resource. In Anna, this could be expressed by the axiom:

```
--| axiom
--|     for all R : Resource =>
--|         Resources'State =
--|         Resources'State[Allocate(R);Deallocate(R)];
```

To see how the package would respond if this axiom were added, we may load the original (axiom-less) package, and transform the state with the "simplified notation" form of the axiom (see [2, §7.8.1]), universally quantifying over the Resources state type:

```
> S (Update state)
--> for all Resource_State : Resources'Type =>
-->   for all R : Resource =>
-->     Resource_State =
-->     Resource_State[Allocate(R);Deallocate(R)];
-->
```

4.4.3 Execute Subprograms — 'x'

The user is prompted for one or more subprogram calls. If the subprogram is a procedure, the call is given just as it would appear in Ada code:

```
> x (Execute subprograms)
--> Set_Output( Output_File );
-->
```

If the subprogram is a function, it may be assigned to a variable, e.g. assuming we have declared a variable T:

```
> x (Execute subprograms)
--> T := Top(S);
-->
```

in which case the function is executed as described below, and in addition to any transformations which occur because of out-state annotations, the expression

T = Top(S)

is TRUE in the new state.

Alternatively, it can be given as simply the call:

```
> x (Execute subprograms)
--> Top(S);
-->
```

in which case the subprogram is executed as described below, and then the value, if any, the function would return is displayed. This kind of subprogram call is not supported if static semantic correctness of input is checked (see Section 4.6.1), since the input is not a semantically legal Ada statement.

Symbolic execution of a subprogram occurs in three stages: checking of in-annotations, checking of exception annotations, and transformation of the computation state using out-annotations:

1. Each in-annotation is queried. If any in-annotation is FALSE or INCONSISTENT, the user is notified and execution of the subprogram halts without a state change. If an in-state annotation is UNKNOWN, the user is prompted as to whether or not execution should be continued.

2. The strong propagation annotations are queried. For all such annotations that are UNKNOWN, a warning message is printed out. For all such annotations that are TRUE, the exceptions that would be raised are sent to output (raising of more than one exception, while impossible in Ada, can occur in Anna Package Specification Analysis if two strong propagation annotations are both TRUE), and these exceptions form a set of raised exceptions.

3. A state transformation occurs. If no exceptions have been raised, the transformation is the set of all out-state annotations. If one or more exceptions have been raised, the set of transformation statements is initially empty. Each weak propagation annotation is examined; if it has a specified exception in the set of raised exceptions, its condition is added to the transformation. In either case, the final transformation set may be empty.

   The call itself (subprogram name plus actual parameters) is appended to the list of previous subprograms executed, to represent the current package state. This completes subprogram execution.

4.5 Logging and Scripting Commands

A log file is a record of the input and output of part of a specification analysis session. Several log files may be open at one time; input and output is sent to all open log files. The most recently opened log file is always closed first—if the user opens log files L1, L2, and L3 in that order, they will be closed in the order L3, L2, L1.

A script file is in a format which may be interpreted by the Analyzer to perform a series of commands automatically, without any user-interaction until the end of the file is reached. An unaltered log file will always be a legal script file, or a script file may be user generated, as long as it is formatted correctly.

The format must be such that it contains zero or more lines beginning (at column 1) with a command prompt ("> ", greater-than and a space) followed by a command character; characters past this column are ignored. If the command takes further input, a line or lines beginning with the input prompt ("--> ", two dashes, greater-than and a space) must appear, along with the required input, after the command and before the next command; for Anna input, a line containing only the input prompt must follow the Anna input to signify its end, just as when entering Anna input interactively.

Script files may also be opened in a nested manner; that is, a script file may itself open a script file. The Analyzer always reads from the most recently opened script file. When the end of a script file is reached, it

is automatically closed and the next most recently opened script file is used for input. When there are no more script files to be read from, control returns to the user interactively.

4.5.1 Begin Log File — 'l'

This command prompts the user for a file name. If the file does not exist, it is created; if it does exist, the original file is destroyed. The file becomes the most recently opened log file.

4.5.2 Close Log File — 'c'

This takes no input. It closes the most recently opened log file.

4.5.3 Execute Script File — 's'

The user is prompted for a script file name. If the file is a readable, legal script file, the Analyzer then begins processing commands and input from the file, until the end of file is reached. The file is then closed, and control returns to the script file which invoked it, or to the user.

4.5.4 Assertion File Test — 'a'

An assertion file has exactly the same format as a script file. The difference is that in an assertion file, the answer to all Boolean Query commands is assumed to be TRUE. As an assertion file executes, it monitors the returned answers to all query ('q') commands. If the answer to one is anything but TRUE, it immediately halts, closes all opened assertion and script files, and returns control to the user, who may then examine the state to deduce why the result was not TRUE.

On Unix systems, assertion files may be given on the command line to automate testing of packages. Details are given in Appendix D.

4.6 Other Commands

4.6.1 Set Options — 'O'

Specification analysis is highly computation intensive. This is complicated by the many different ways of writing specifications: a method of analysis which is fast and helpful for one kind of specification may be slow and less robust for other kinds. As opposed to restricting the Analyzer to supporting only certain kinds of specifications, it has been made as general as possible, and provided with options so that a particular session may be tuned to the kind of specification being analyzed.

For instance, if the package specification describes an abstract data type, and requires no internal state, it is very much to the user's advantage to set the Trivial States option accordingly. Or, while confidence that the computation state is logically consistent is crucial in giving confidence that the specification behaves as expected, consistency checking takes a great deal of time. If the user is fairly confident that the specification is consistent, he will probably want to check consistency manually and only after certain state transformations in which he has less confidence.

When the Set Options command is invoked, for each option kind the available settings and the current setting are printed. Then the user is prompted for a new setting; this is always a single digit, 0-9, and no carriage return is required. To keep an option the same, enter a carriage return.

Each option and its use is described below.

Single Value Search If the state being examined is inconsistent, it is possible for an expression to have different values in different consistent subsets of the state. Because the alternative values of an expression may be useful in determining why a state is inconsistent, the Value Search command may optionally search the state for all values of an expression. If this option is set to TRUE, once a value for the expression is found, the Analyzer prints it and stops looking. If the state is known to be consistent, then every expression must have at most one value (due to the uniqueness property of Ada literals), and this is the best and fastest mode. If FALSE, the state is searched for all values in all subsets of the state which are consistent; they are printed as they are found. The default is TRUE.

Semantics When TRUE, this enables static semantic checking of all packages loaded, and of interactive Anna input. After setting this option to TRUE, the Initialize command ('i') should be given, and all packages should be reloaded. The default is TRUE.

Depth Limit The "depth" of an inference is the greatest number of rules used to derive it. For instance, if the rules

for all I : Integer => I > 0 -> C(I)
X = 10

in the current state are used to derive

C(X)

the depth reached was 2. Setting this option to a positive number limits to that number the maximum depth explored in trying to prove statements or deduce the value of expressions. When set to 0, no depth limit is enforced. The default is No Limit.

Trivial Package States When TRUE, packages loaded are assumed to have trivial states; that is, all states of the package are assumed equal to the Initial state. This simplifies some theorem proving, and the representation of the package is also simpler. Usually the time required for queries and value searches decreases. If the package contains annotations which cannot hold for a trivial state package, they may be detected as a state inconsistency. The default is FALSE.

State Test If TRUE, the state consistency test command ('t') will be invoked automatically after each computation state transformation (i.e., after any State Manipulation command). The default is FALSE.

Save Corollaries Deducing the result of a query or the value of an expression may be a complex process taking a great deal of time. Also, it often happens that a boolean expression whose truth value has been deduced is used later in deducing the truth value of another expression. Normally the Analyzer would have to prove the original expression all over again. This option allows users to save the results of the Query and Value Search commands automatically. The default is FALSE.

When this option is set to TRUE, and a Query is given which has the result TRUE, the deduced statement is added directly to the computation state, so that it may be used directly in future deductions. When a Value Search command is given and a value is found, an expression equating the given expression with the resulting value is added to the state. For instance:

```
> v (Value of expression)
--> I;
-->

23
```

After this command is given, the expression

I = 23

is added to the state if the Save Corollaries option is TRUE.

This does not cause a state transformation; the computation state number remains the same. This is because no new information is being added. Instead, a corollary of the state is being stated explicitly.

Quantified Expression Query Kind Setting this option automates the method chosen for evaluating quantified expressions when using the Boolean Query command. Methods of evaluating quantified expressions are discussed in Appendix C. The default is None.

None. For every quantified variable in expressions which are queried, the user is asked interactively for the method of evaluation.

Check Objects Only. All quantified expressions are evaluated using the Check Objects Only method.

Objects Define Domain. All quantified expressions are evaluated using the Objects Define Domain method.

Verbose If this option is set to TRUE, intermediate steps for several commands are shown. For instance, in executing a subprogram with verbosity the testing of each in-axiom and propagation annotation will be noted, as will the updating process. The default is FALSE.

4.6.2 Enter Windowing Mode — 'w'

When running the Analyzer under X-Windows, a window-oriented user interface is available. It features a main window with pull-down menus for entering commands, and "buttons" for setting options. When a file containing one or more packages is loaded for analysis, a second window is activated showing the text of the file. The appropriate lines of the file are highlighted where, in the line-oriented version, source text is displayed. User input is entered in the main window, in the usual format (prompted by "-->", and terminated by an empty input line).

There are some differences in functionality in the window-oriented version, as follows:

- There is no "Set Options" command. Options are set by pressing buttons at the top of the main window.
- The quantified expression query kind is fixed as "Objects Define Domain" (see Appendix C) upon entering windowing mode, and can not be reset; there is no corresponding button for this option.
- Only one file may be analyzed at a time; however, the file may contain multiple packages.

4.6.3 Help — 'h' or '?'

Currently, this command prints out the list of single-character commands, along with a one-line description of each.

4.6.4 Quit — 'Q'

Exits the Analyzer.

Chapter 5

Implicit Knowledge

5.1 Package Standard

The Analyzer has some knowledge of the package Standard, defined in [8, §C]. This knowledge is as follows.

Types Boolean, Integer, Character, Natural, Positive, and String are defined.

For relational operations, all relations are represented internally as equalities, less-than relations, and conjunctions and disjunctions of the two (e.g. $a >= b$ is represented as a = b **or** b < a). The implicit axioms for equality ("=" given in [2, §7.8.2]) are supported. The strict partial order axioms (transitivity and irreflexivity) for the less-than relation "<" are supported.

Logical operations are all defined. All Integer type operations are defined, and will yield the expected function results if passed expressions which have a value in the given state. Axioms of the integer operations (commutativity, associativity, etc.) are not implicit. Thus while it is easy to prove

A + B = B + A

when A and B have deducible values, deduction of commutativity in general is not supported.

No other definitions in package Standard are supported.

5.2 Frame Axiom

A computation state is comprised of many entities with values which may be different in different states, such as variables and function calls. When specifying a state transformation, it is convenient to describe only the changes to the state, and ignore entities in the state whose values are unchanged; intuitively, if a state transformation does not refer to a state entity, we may assume that its value has not changed. This implicit assumption is known as the Frame Axiom. Since variables and function calls are two fundamentally different entities in this model (variables existing in the outer computation state, and indexed by numbers; function call values existing in a particular package state, and indexed by a list of subprogram calls), two different schemas are needed.

5.2.1 The Frame Axiom for variables

Consider a set of Anna expressions which transform the state S(i-1) into S(i), where i is a positive number. Associate with this transformation a set Used(i), which is the set of variables that are assumed to have been changed in value by the ith transformation. Then we may express the Frame Axiom as an axiom schema which applies to each variable declared. This schema is as follows:

For each variable v,
$(\forall$ *positive* $i)$ *if* $v \notin Used(i)$ *then* $v'i = v'(i-1)$

The Analyzer builds Used(i) by examining instances of variables in the set of transformation expressions. If some instance of a variable is not modified by **in** (meaning the instance refers to the value of the variable after the transformation), it is an element of Used(i). A variable occurring in the transformation which has every instance modified by **in** will not be in Used(i). For example, if the transformation to computation state *i* consists of the single Anna expression:

```
--| X = in Y;
```

then X will be the only element of Used(i); so from the schema we can infer that in state *i*,

```
y'(i) = y'(i-1)
```

This matches the intuition of most users that the transformation implies that the value of X has changed, but not the value of Y.

5.2.2 The Frame Axiom for function calls

Now consider a set of Anna expressions which transform the state of a package P from the package state P'State to the state P'State[S] by execution of subprogram call S. Associate with this transformation a set

```
P_Used(P'State[S])
```

(that is, P_Used is a collection of sets indexed by states of P), which contains instances of function calls. Then we may express the Frame Axiom as an axiom schema for each legal function call $F(x_1, \ldots, x_n)$ for each n-ary function F, and each expression x_i (note that x_i is not a value but an Ada *expression*) in package P. This schema is:

For each $n-$ *ary function* F *in* P,
$(\forall x_1, \ldots, x_n)$ *if* $f(x_1, \ldots, x_n) \notin P\_Used(P'State[S])$
then $P'State[S].F(x_1, \ldots, x_n) = P'State.F(in(x_1), \ldots, in(x_n)))$

The meaning of $in(x_i)$ is analogous to its meaning in Anna: if expression x_i contains a variable, use the value of the variable in the previous computation state; if x_i contains another call to a subprogram of package P relative to P'State[S], use the value of the function in state P'State with **in** applied to *its* actual parameters.

P_Used is built only when the state of some package changes; this can occur only when performing the Execute Subprograms command. P_Used is built in a manner similar to the Used set. When executing a subprogram symbolically, the computation state transformation may contain calls to functions declared in the same package as the subprogram begin executed (we assume calls to other packages do not change the states of those packages). For each such call, if it is in an out- or result-annotation and is not modified by **in**, the call is an element of P_Used. As with variables, we may think of this intuitively as follows: if the user makes an assertion about the value of a package function in the out-state of a subprogram execution, the user probably means to imply that its value has changed by executing the subprogram.

For example, if in a package Direct_IO we have a subprogram such as:

```
procedure Write( F : File;
                 P : Position;
                 D : Data );
--| where
--|     Is_Open(F),
--|     out( Data_At(F,P) = D );
```

and we execute this subprogram in the initial state with the following call:

```
Write( F1, 3, D1 );
```

then the function call Data_At(F1,3) will be a member of the set

```
Direct_IO_Used(Direct_IO'Initial[Write(F1,3,D1)])
```

whereas the other function call referred to in the subprogram annotations, Is_Open(F1), will not be a member of that set since it did not occur in an **out** annotation. Therefore, for example, the frame axiom will apply to the Is_Open call; the following statement will hold:

```
Direct_IO'Initial[Write(F1,3,D1)].Is_Open(F1) = Direct_IO'Initial.Is_Open(F1)
```

In fact, it would apply to every legal function call in the package except the call Data_At(F1,3).

A function call is also in P_Used if it is used in an axiom or result annotation relative to a state matching the subprogram execution. If a package contains the axiom

```
--| axiom for all R : Resource =>
--|       not Resources'State[Allocate(R)].Resource_Available(R);
```

This axiom expresses that the value of Resource_Available for a particular resource will be FALSE immediately after an Allocate of that resource. The sets Resources_Used are indexed by a package state; for every state whose last call was an Allocate of some resource R, its associated Used set will contain Resource_Available(R). Examples of sets containing Resource_Available(Mag_Tape) are:

```
Resources_Used(Resources'Initial[Allocate(Mag_Tape)])
Resources_Used(Resources'Initial[Allocate(Printer1);Allocate(Printer2);Allocate(Mag_Tape)])
Resources_Used(Resources'Initial[...;Deallocate(Mag_Tape);Allocate(Mag_Tape)])
```

and so on. This corresponds to the user's intuition that the axiom above "forces" the value of Resource_Available to be FALSE in all applicable situations.

Finally, if a function has a result annotation, all possible calls to the function will be elements of all P_Used sets; by writing a result annotation, the user is implying that the value of a function is reflected in the result annotation no matter what state of the package the function is called relative to. If the following declaration is given in a package Lists:

```
function Member( E : Elem; L : List ) return Boolean;
--| where return for all L2 : List =>
--|     if L = Insert(E,L2) then TRUE
--|     else Member(E,L2) end if;
```

the user is making an assertion about the value of Member in every state of package Lists, so every possible function call to Member is an element of all sets in the collection Lists_Used.

Bibliography

[1] David C. Luckham. *Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs.* Texts and Monographs in Computer Science. Springer-Verlag, October, 1990.

[2] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. *ANNA, A Language for Annotating Ada Programs*, volume 260 of *Lecture Notes in Computer Science.* Springer-Verlag, 1987.

[3] John J. Kenney and Walter Mann (eds.). Anna package specification: Case studies. Technical Report CSL-TR-91-496, Stanford University, October 1991.

[4] S. Sankar. *Automatic Runtime Consistency Checking and Debugging of Formally Specified Programs.* PhD thesis, Stanford University, August 1989. Also Stanford University Department of Computer Science Technical Report No. STAN–CS–89–1282, and Computer Systems Laboratory Technical Report No. CSL–TR–89–391.

[5] R. Neff. *Ada/Anna Package Specification Analysis.* PhD thesis, Stanford University, December 1989. Also Stanford University Computer Systems Laboratory Technical Report No. CSL–TR–89–406.

[6] W. Mann. Representation of an Anna subset in predicate logic for specification analysis. Unpublished Technical Report.

[7] Geoffrey O. Mendal et al. *The Anna-I User's Guide and Installation Manual.* Stanford University, Computer Systems Lab, ERL 456, Stanford, California, version 1.5 edition, December 1992.

[8] US Department of Defense, US Government Printing Office. *The Ada Programming Language Reference Manual*, February 1983. ANSI/MIL-STD-1815A-1983.

Appendix A

Sample Scenario

What follows is a simple scenario using the Analyzer on a small Set package. The annotated package itself is shown in Figure A.1. Note that we are analyzing the generic package itself, and not an instantiation of it (see Appendix B.1.12); briefly, the type Elem is treated as a private type, in the same way as type Set is. Also, while an implementation is given for the private type Set, the private part of a package is ignored by the Analyzer. Declarations in the private part are included only to insure Ada semantic correctness of the unit.

```
generic
   type Elem is private;
package Sets is
   type Set is private;

   function Empty return Set;
   function Insert( E : Elem;  S : Set ) return Set;

   function Cardinality( S : Set ) return Natural;
   function Member( E : Elem; S : Set ) return Boolean;

--| axiom Cardinality(Empty) = 0;

--| axiom for all E : Elem => not Member( E, Empty );

--| axiom
--|     for all S : Set; E : Elem =>
--|         Cardinality(Insert(E,S)) = Cardinality(S) + 1,
--|         Member( E, Insert(E,S) );

private
   type Set is ...;
end Sets;
```

Figure A.1: The Set Package

This package is an example of using Anna to specify an abstract data type—namely, a set—and standard

operations performed on the type. Two of the functions, Empty and Insert, operate as generators of the Set type and the other two, Cardinality and Member, operate as observers. Package axioms relate the semantics of the generator and observer functions.

One methodology for debugging this kind of package using Specification Analysis is to build different values of the abstract type (Set) using the generator functions and to store them in interactively declared variables. Then the user examines the values of observer functions on the interactive variables; the value an observer has is deduced from the axioms using theorem proving techniques internal to the Analyzer. If the observer values are not consistent with what the user expected, he tries to isolate where the difference arose by focusing on the annotations relevant to the observer function.

This methodology is applied in detail below. A trace of a debugging session (boxed) is interspersed below with commentary. In the trace itself, user input is shown in **boldface**, and Analyzer output in Roman.

First, we will want to save this session for further reference, so we open a new log file "set_session." It is clearly the intention of the user that the functions of this package are "pure;" no data internal to the package body is needed to support the functions. The Analyzer has an option (see Appendix 4.6.1) which may be set so that the package is assumed to have only pure functions and no internal state; it is called the Trivial State option. So the Trivial State option is first set to TRUE; no other options are changed. Then the computation state is initialized using the 'i' command, and the package is loaded; assume the package given in Figure A.1 resides in the file "set.anna".

```
> l (Begin log file)
Enter File Name:
--> set_session

> O (Set options)
Type <cr> to keep an option the same.
Single Value Search (0=False(All values), 1=True(Single value))[True]
-->
Semantics (0=False, 1=True)[False]
-->
Depth Limit (0=No Limit)[No Limit]
-->
Trivial Package States (0=False, 1=True)[False]
--> 1
State Check (0=False, 1=True)[False]
-->
Save Corollaries (0=False, 1=True)[False]
-->
Quantifier Query Method [Interactive]
-->
Verbose (0=False, 1=True)[False]
-->

> i (Initialize computational state)

> p (Load package specifications)
Enter File Name:
--> set
```

Next some useful variables are declared. S1 is a set which is initially empty. E1 and E2 are elements;

since Elem is private and they are not given initial values, they are both considered equal to the Anna expression Elem'Initial. Once we have declared them, it is useful to transform the state such that E1 and E2 are arbitrarily valued, but definitely not equal to each other; then statements we prove about them will be TRUE for any two distinct Elem values. So the state is updated with an expression which implies exactly that situation. Finally the values of the observers with respect to S1 are examined, to see if they have the expected values.

```
> o (Declare objects)
Enter Code (<CR> to end):
--> S1 : Set := Empty;
--> E1, E2 : Elem;
-->

> S (Update state)
Enter Code (<CR> to end):
--> E1 /= E2;
-->

> v (Value of expression)
Enter Code (<CR> to end):
--> Cardinality(S1), Member(E1,S1);
-->
0

FALSE
```

The observers do indeed have the expected values. Now it is up to the user to generate more complex values of the type and examine their observer values. For instance, suppose we suspect that the package may not specify as the user intended multiple insertions of the same element into a given set. Then an execution sequence is given which inserts element E1 into S1 twice, and again observer values are checked.

```
> x (Execute subprograms)
Enter Code (<CR> to end):
--> S1 := Insert(E1,S1);
--> S1 := Insert(E1,S1);
-->

> v (Value of expression)
Enter Code (<CR> to end):
--> Cardinality(S1), Member(E1,S1);
-->
2

TRUE
```

While the value for the Member call is the expected one, most users would agree that set cardinality should ignore multiple insertions of the same element, and the cardinality of S1 should be 1. In fact, a "correct" implementation of Cardinality (one which returned 1) would violate the given Cardinality axioms. Hence the annotations of the package do not express the intended semantics of Cardinality. To see where

the problem lies, we examine those annotations which determine the value of Cardinality; two axioms do so:

```
--| axiom Cardinality(Empty) = 0;
--| axiom
--|     for all S : Set; E : Elem =>
--|         Cardinality(Insert(E,S)) = Cardinality(S) + 1;
```

We saw when variable S1 was initialized that the first axiom implied the correct value for Cardinality of an empty set, and so the problem must lie in the second axiom. A correct version of this axiom which handles repeated insertions is given below.

```
--| axiom
--|     for all S : Set; E : Elem =>
--|         Cardinality(Insert(E,S)) = if Member(E,S) then Cardinality(S)
--|                                    else Cardinality(S) + 1 end if;
```

We can continue to examine the validity of the Member function in this session. The element E2 is inserted into the existing set, and Member is tested for both elements.

```
> x (Execute subprograms)
Enter Code (<CR> to end):
--> S1 := Insert(E2,S1);
-->

> v (Value of expression)
Enter Code (<CR> to end):
--> Member(E2,S1), Member(E1,S1);
TRUE

Unable to deduce a value for this expression.
```

We expected E2 to be a member of the set, and this turned out to be the case. However, we also expected E1 still to be a member, but this is not provable. Somehow the membership of E1 has been "forgotten." Again we narrow down our search to the applicable annotations:

```
--| axiom for all E : Elem => not Member( E, Empty );
--| axiom
--|     for all S : Set; E : Elem => Member( E, Insert(E,S) );
```

In the same way as before, we see that the second axiom does not reflect the intended semantics of Member; it implies membership only for the element most recently inserted. Unlike the previous problem, a correct implementation would not violate this axiom. It is simply not complete enough, providing too little information to deduce membership of all inserted elements. A more complete axiom is given below.

```
--| axiom
--|     for all S : Set; E1,E2 : Elem =>
--|         Member( E1, Insert(E2,S) ) <-> E1 = E2 or Member(E1,S);
```

At this point we wish to end the session and rewrite the package with corrected axioms. The log file is closed.

```
> c (Close log file)
Closed set_session
```

Once the changes are made, we start a new session using the log file as a script file; this will give all the commands of the previous session exactly as we gave them. The observers should now exhibit the expected

values.

```
> s (Execute script file)
Enter File Name:
--> set_session
⋮
```

After this experiment, we may want to try other generated Set values and examine the observers, until we are satisfied that the annotations express the intended semantics of sets.

Appendix B

Description of the Supported Subset and Limitations

B.1 The Supported Subset

B.1.1 Introduction

What follows describes the subset of Ada and Anna that is supported in the current Analyzer. The description has been organized in sections which follow the chapters of the Ada/Anna manuals [8, 2].

When the Ada/Anna Semantic Checker is used to insure static semantic correctness of all input (see Section 4.6.1), some features of the Analyzer are not supported. These are noted where applicable in this document. Otherwise, the Analyzer assumes that all input, both files and interactive, is semantically legal.

Where the semantic checking option is not used, some minimal checking is still performed. Error messages are given when certain obvious semantic errors are detected (asking to execute a subprogram that does not exist, for example); in addition, uses of object, type and function names are checked to see if the names have been defined. Static type checking is not performed when the semantic checking option is not set.

B.1.2 Lexical Elements

Supported for discrete types. Pragmas are ignored.

B.1.3 (Annotations of) Declarations and Types

Object declarations, named number declarations and object annotations are supported.

Type declarations, subtype declarations, and subtype annotations are supported for discrete types and private types only.

The DEFINED attribute is supported for objects. The INITIAL attribute is supported for types.

Subtypes and the BASE attribute are supported for discrete and private types.

Derived types are supported for discrete and private types.

All discrete types are supported except where their literals are identifiers or characters which are already used as literals. (e.g., Character Enumeration types, or user-defined enumeration types with overloaded literals).

The FIRST and LAST attributes are supported. All discrete type operations are supported except the attribute WIDTH.

Floating point, Fixed point, array, record, and access types, and annotations for these types, are *not* supported.

The type String is supported as a special case. String values may be examined with the Value Search option, and equality is defined for the type. However, no other string operations are defined.

B.1.4 Names and Expressions (in Annotations)

Attributes and literals are supported for discrete types. Named annotations are allowed, but the name is ignored.

All Ada and Anna operators for discrete types are supported (membership, implication operators, logical operators, short circuit operators, and relational operators). Ada membership (**in**) and Anna membership (**isin**) are supported.

Type conversions and qualified expressions are supported for discrete and private types only.

Universal_Integer is supported, Universal_Real is *not*.

Quantifiers are supported for discrete and private types. Evaluating a quantified expression is accomplished by one of three methods, which are described in Appendix C.

Conditional expressions and modifiers are supported. Definedness of expressions is *not* supported, except for objects and function calls. For definedness of a function call, currently the in-annotations and strong propagation annotations are not tested; that is, a function call is considered defined if its actual parameters satisfy the Ada and Anna type constraints of the formals, even if an in-annotation is violated. All other expressions are assumed to be defined.

B.1.5 Statements

The only statements that may be executed in Specification Analysis are subprogram executions and assignments of function calls to variables; they are supported. Using the semantic checking option, only semantically legal subprogram calls are allowed. Without this option, the user may also enter simply a function call, not assigned to any object (as described in Section 4.4.3).

B.1.6 (Annotation of) Subprograms

All formal parameter modes are supported. For subprogram calls, default and named actual parameters are supported.

All subprogram annotations are supported (in, out, result, propagation annotations and object constraints).

Overloading of subprograms and operators is *not* supported.

Subprogram attribute NEW_STATE is supported. OUT is *not* supported.

B.1.7 (Annotation of) Packages

Package specifications with no nested packages or generic instantiations are supported.

Private and limited private types without discriminant parts are supported. Deferred constant declarations are supported.

The private part of a package is ignored.

The private type attributes, BASE, SIZE, CONSTRAINED, and ADDRESS are *not* implemented.

Because overloading is not supported, neither is redefining "=" for limited types. The operation "=", when used with limited types, is the default defined by Anna; it obeys the standard axioms of equality.

Successor package states, relative function calls, and annotations on state types are supported.

Axiomatic annotations are supported. The implicit axioms of equality (reflexivity, symmetry, transitivity, substitution, and independence) are supported.

Package consistency can be checked manually or automatically every time the package state changes, and reports if an inconsistency has occurred.

B.1.8 Visibility

Renaming of declarations is *not* supported.

Package Standard is implemented for all declarations involving discrete types except nested package ASCII.

B.1.9 Tasking

Not applicable.

B.1.10 Program Structure

with clauses are allowed, but are ignored; semantic correctness of references to external packages is assumed. **use** clauses are *not* supported.

B.1.11 (Annotation of) Exceptions

Exception declarations are supported.

The only predefined exceptions which could be raised in a computation during Specification Analysis (that is, not as a result of a propagation annotation) are Constraint_Error or Numeric_Error. They are supported.

Propagation annotations are supported.

Suppressing of checks is *not* supported.

B.1.12 (Annotation of) Generic Units

When the semantic checking option is used, generic packages are *not* supported.

When semantic checking is not used, generic packages may be analyzed by themselves, without being instantiated. The generic parameters act as though they were declared in the visible part of a non-generic package: generic formal types act like private types, generic formal **in** objects act like constants, generic formal **in out** objects act like specification variables, generic formal subprograms act like normal specification subprograms. All generic formals are supported for discrete types. This includes formal discrete and formal integer types; but use of attributes FIRST or LAST is *not* supported for these types. Annotations of generic formal parameters are supported; they act like object annotations.

Instantiation of generic packages is *not* supported.

B.1.13 Implementation Dependent Features

Not applicable.

B.1.14 Input-Output

Not applicable.

B.2 Limitations

The Analyzer is a young and not-so-robust tool. Its current release is intended as an exercise to see if/how people will use it, and what problems they encounter.

While most of the theory is well-understood, the problems of writing a decent theorem prover must be faced. The current prover is slow and imperfect; you may find problems with it. Please let us know.

One particular theoretical problem arises when using interactively declared variables in a non-trivial state package. The frame axioms for variables and functions sometimes interact in strange ways. If you think you are running into this problem, contact the Stanford Anna Team.

Appendix C

Methods of Evaluating Quantified Expressions

When invoking the Boolean Query command on an Anna expression containing a subexpression which is existentially or universally quantified, the expression is not proved for the entire domain of the quantified variable type—this is, in general, a very difficult problem. Instead, the Analyzer uses substitution schemes to test the expression using a subset of the type domain of each quantified variable in the expression.

In each method, a set of Ada literals or objects is associated with each quantified variable. The Analyzer then creates a set of expressions without quantifiers, by substituting an appropriate literal or object for the quantified variable instances in the expression. There is one expression for each combination of substitutions from the domain sets; each expression is queried, and the set of queries yields a set of boolean results. Each method interprets the result of the quantified expression in terms of the set of results.

There are three different substitution methods. They are described below.

C.1 Discrete Range

The first method, *Discrete Range*, can only be used on quantified variables of a discrete type. It prompts the user for a range over which to evaluate the expression. The set of substitutions for the quantified variable is then the set of literals in the given range.

- for universal quantifiers, the expression is:
  - TRUE if all queries of the set of substituted expressions are TRUE,
  - FALSE if at least one query is FALSE, and
  - UNKNOWN otherwise;
- for existential quantifiers, the expression is:
  - TRUE if at least one query is TRUE,
  - FALSE if all queries are FALSE, and
  - UNKNOWN otherwise.

This method is best used when the extent of the type domain itself is being tested; for instance to test that a subtype annotation restricts the domain as expected. Often in such cases, a small subset of boundary cases are all that need be tested.

For example, suppose we are writing a square root function for natural numbers. To test that it behaves as expected, we might query:

for all N : Natural => N <= SQRT(N)**2 <= N+1;

While in general we would need to show that this query holds for all natural values N, we would probably be satisfied by trying a few values, say 0 to 3. Using the Discrete Range method with the range 0..3, this would mean that the set of substitution expressions would be:

0 <= SQRT(0)**2 <= 0+1;
1 <= SQRT(1)**2 <= 1+1;
2 <= SQRT(2)**2 <= 2+1;
3 <= SQRT(3)**2 <= 3+1;

If each expression in this set is provably TRUE; the returned answer is TRUE. If any expression is provably FALSE, the returned answer is FALSE. If some are TRUE and some are UNKNOWN, the returned answer is UNKNOWN.

C.2 Check Objects Only

In the second method, *Check Objects Only*, the substitution set consists of the names of objects (variables and constants) of the quantified variable's type which have been declared so far.

- for universal quantifiers, the expression is:
  - FALSE if at least one query is FALSE, and
  - UNKNOWN otherwise;
- for existential quantifiers, the expression is:
  - TRUE if at least one query is TRUE, and
  - UNKNOWN otherwise.

This method assumes that there are values in the type domain which are not represented by any existing object—most likely the case for complex types. This is a conservative approach the main function of which is to search for counterexamples based on the known values of the type.

For example, suppose we want to test the Singleton function of a Set package. So far, we have interactively declared three Set type objects: S1, S2, S2. To prove the expression:

for all S : Set => Singleton(S) -> Cardinality(S) = 1;

this method would prove the set of expressions:

Singleton(S1) -> Cardinality(S1) = 1;
Singleton(S2) -> Cardinality(S2) = 1;
Singleton(S3) -> Cardinality(S3) = 1;

Suppose one of these expressions were provably FALSE. Then the original expression is FALSE—we have found a counterexample. Suppose all the expressions were provably TRUE. Then the answer is UNKNOWN, since we still cannot prove the expression holds for every value in the Set domain.

C.3 Objects Define Domain

In the last method, *Objects Define Domain*, the substitution set also consists of the names of objects of the quantified variable's type which have been declared so far. The difference is how the results of the set of queries is interpreted:

- for universal quantifiers, the expression is:

  - TRUE if all queries are TRUE,
  - FALSE if any query is FALSE, and
  - UNKNOWN otherwise;

- for existential quantifiers, the expression is:
  - TRUE if at least one query is TRUE,
  - FALSE if all queries are FALSE, and
  - UNKNOWN otherwise.

This method assumes that the entire domain is represented by values of all objects that have been declared. That does not necessarily need to be an accurate view; rather, it can simply show how the type reacts "so far." Both Check Objects Only and Objects Define Domain are most useful for private types, where no domain is explicitly determinable.

For example, returning to the example for Check Objects Only, suppose one of the expressions is provably FALSE. Then the original expression is still FALSE. Suppose that all of the expressions are TRUE. Then the original expression is now provably TRUE; *as far as we know*, the original expression holds for the entire domain.

Appendix D

Invoking the Analyzer From Unix

If the Analyzer and the rest of the Anna tools have been correctly installed (see [7] for details), the Analyzer is invoked on Unix systems with the command:

```
/usr/anna/bin/span ([option]* [file_name]*)*
```

That is, the name of the Analyzer executable file is followed by any combination of command-line options (preceded by a dash character '-') and file names. Many command-line options in turn set Analyzer options described in Section 4.6.1. `option` can be:

- *-find_first_value* — Sets the Value Search option to TRUE. This is the default.
- *-find_all_values* — Sets the Value Search option to FALSE.
- *-semantics* — Sets the Semantics option to TRUE. This is the default.
- *-no_semantics* — Sets the Semantics option to FALSE.
- *-test_state* — Sets the State Test option to TRUE.
- *-no_test_state* — Sets the State Test option to FALSE. This is the default.
- *-trivial_state* — Sets the Trivial State option to TRUE. This is the default.
- *-no_trivial_state* — Sets the Trivial State option to FALSE.
- *-save_corollaries* — Sets the Save Corollaries option to TRUE.
- *-no_save_corollaries* — Sets the Save Corollaries option to FALSE. This is the default.
- *-method method_kind* — Sets the Quantified Expression Query Kind option. *method_kind* is either "none," "check_objects" or "objects_define_domain." "none" is the default.
- *-verbose* — Sets Verbose option to TRUE.
- *-no_verbose* — Sets Verbose option to FALSE. This is the default.
- *-windowing* — The Analyzer will use the X-Windows interface.
- *-assertion_files* — File names following this option in the command line are interpreted as assertion files; all file names preceding it are interpreted as load files. If this option is not given, all file names are interpreted as load files.

  If any assertion files are given, a special form of execution occurs: the load files are loaded, and the assertion files are tested in the order given. Any error in the assertion files (loading error, syntax error, or non-TRUE assertions) causes the execution to go immediately into normal interactive mode, so that the error may be examined. If no errors are found, the Analyzer terminates after the last assertion file has been tested. This option can be used to automate testing of packages.

`file_name` is the name of either a load file or an assertion file, as described above for option "-assertion_files." A file name has no leading '-'. As with the Load Packages command, the ".anna" extension need not be given for load file parameters.

Under Unix, **ctrl**-C will cancel any command using deductive proof (Value Search, Boolean Query, Consistency Test, or Execute Subprograms). **ctrl**-C elsewhere in the Analyzer has no effect.

The file `/anna/spec_analyzer/span_ui/READ_ME` lists specific information on using the X-Windows interface, for example X default options for the Analyzer.